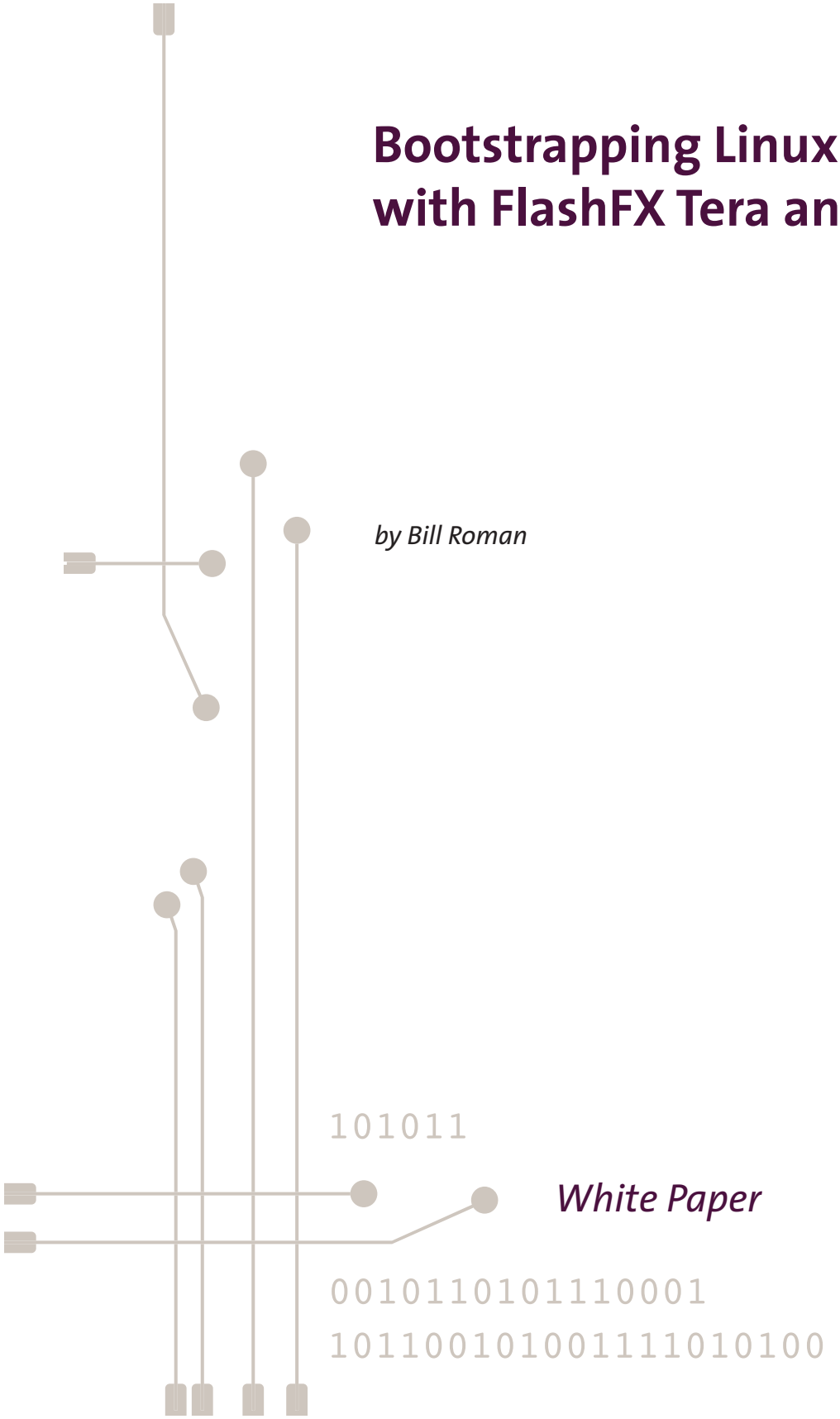
Bootstrapping Linux from NAND Flash with FlashFX Tera and Reliance Nitro

by Bill Roman



White Paper

Copyright © 2006-2010 Datalight, Inc. All rights reserved. Printed in USA. DATALIGHT, Datalight, the Datalight Logo, FlashFX, FlashFX Pro, Reliance, ROM-DOS, 4GR, One-Boot, One-Boot+File, and Sockets are trademarks or registered trademarks of Datalight, Inc. All other product names are trademarks of their respective holders. Specification and price change privileges reserved.

Bootstrapping Linux from NAND Flash with FlashFX Tera and Reliance Nitro

Author: Bill Roman

Table of Contents

Bootstrapping Linux from NAND Flash with FlashFX Tera and Reliance Nitro 1
- Introduction 2
- Overview of Booting Linux 2
  - Hardware initialization 3
  - Loading the kernel 3
  - Mounting the root file system 4
  - Running the init process 5
  - Extending the kernel with loadable modules 5
- Introduction to the Datalight Linux Boot Loader 5
- Analysis of the Boot Loader 6
- The Initial RAM Disk 9
- Finishing the Linux Bootstrap Process 11
- Installing and Using the Boot Loader 11
  - Building the Kernel 12
  - Building FlashFX Tera and Reliance Nitro 12
  - Building the Boot Loader 12
  - Creating the Initial RAM Disk Image 13
  - Installing Using an NFS Root File System 14
  - Installing the Boot Loader 15
  - Using the Boot Loader 16
- Concluding 16
- Appendix A: List of project files (with brief descriptions) 16
- Appendix B: Glossary 17
- Appendix C: Initial RAM disk contents 17

Introduction

This document describes how the Datalight FlashFX® Tera flash media manager and Reliance™ Nitro file system may be used to enable an embedded system to boot Linux from resident flash memory. It refers to a sample project for the LogicPD MX31-lite development board. This project is designed to be easily extended or modified to work with other hardware platforms.

In order to make use of the sample project, you should already have built a kernel and the FlashFX and Reliance kernel modules for the intended target system. Steps for building FlashFX Tera and Reliance Nitro are documented in their respective Development Guides. Building the kernel should be documented in your kernel development system.

This document covers details related to Datalight software, but assumes familiarity with the Linux environment and commonly used tools. Thus, for example, it will tell you to “use fdisk to create partitions on the flash disk,” but will not give detailed directions on the exact commands to issue.

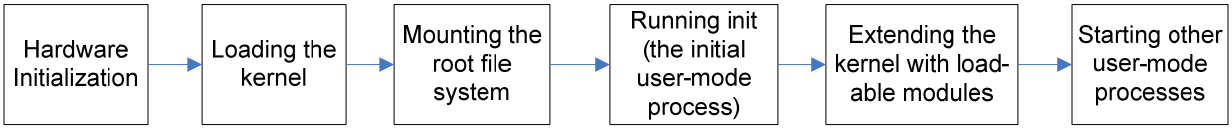
Overview of Booting Linux

The process of getting an embedded system from initial power-on state to fully operational can seem complicated, convoluted, and perhaps a bit arcane. This is especially true when it involves a sophisticated multitasking operating system such as Linux and complex memory such as NAND flash. Linux has developed a versatile set of techniques that accommodate a wide range of system configurations and requirements. FlashFX and Reliance can take advantage of the same methods that are employed for other mass storage and file system drivers. A review of the Linux boot process will help you understand how the Datalight bootstrap program works.

A running Linux system consists of the Linux kernel and a set of user-mode programs managed and serviced by the kernel.

All of these programs reside in one or more file systems. Unlike operating systems that assign a "drive letter" to each file system, Linux (and any UNIX-like operating system) combines all file systems into one tree-structured hierarchy. The very first file system (to which all others are attached) is called the "root" file system. Everything else branches out from there.

The process for starting Linux may be broken down into these major steps:


Hardware Initialization
Loading the kernel
Mounting the root file system
Running init (the initial user-mode process)
Extending the kernel with load-able modules
Starting other user-mode processes

Each of these steps has different ways it can be accomplished, depending on the system configuration and requirements.

Hardware initialization

This part of the process is highly dependent on the hardware platform. It may be necessary to initialize the processor, its memory management hardware, and components like memory controllers and essential I/O interfaces.

In many designs this is accomplished by software residing in NOR flash that is executed automatically after power-on or hardware reset. This software is called the boot loader, and can be a general-purpose monitor program such as U-Boot,[1] or it may be custom-written for the system hardware. Writing startup code to run from NOR flash requires detailed knowledge of the hardware initialization requirements, but is usually straightforward.

Some modern processors have NAND flash interfaces that support reading the first startup code directly from flash. Because only a small amount of code can be loaded this way, the boot loader must be capable of operating the NAND interface in order to read the rest of the startup code. This approach is attractive because it eliminates the need for a separate NOR flash part, but adds some complexity to the initialization process.

Loading the kernel

Once the hardware has been configured and initialized to a usable state, the Linux kernel can be loaded into RAM from permanent storage. It is then started by jumping to its entry point and providing the essential information about the hardware and software configuration. The details of this process vary depending on the processor architecture and the firmware used for initialization.

The Linux kernel image may be stored in a dedicated area of flash memory (either NOR or NAND), or may be accessed through a file system on either flash memory or a hard disk. Loading from NOR is simple. Since NOR flash typically appears to the processor as if it is ordinary memory, loading is just a memory-to-memory copy operation. Accessing NAND flash is more complex because it involves transferring the data as a sequence of fixed-size pages, sending it a command and address for each.

---

[1] http://sourceforge.net/projects/u-boot/

For all the simplicity of this approach, the risk of storing the kernel in a dedicated area of the flash lies in updating the kernel safely. Since the kernel is stored at a fixed location, it's necessary to erase then reprogram that area of the flash. While this is being done, an unexpected interruption will leave the system in an unrecoverable state, without a kernel to load. In addition, if NAND flash is used, the boot loader must handle potential bad blocks and provide error detection and correction. Loading the kernel from a file system allows most of the logic of reliable updates to take advantage of the full capabilities of the operating system, and thus simplifies the actual boot process.

Storing the kernel in a file system can have other advantages too, especially when software updates are considered. Inevitably, features must be added, or bugs must be fixed, and the kernel image must be updated. In a well-designed file system, it's easy to replace an old kernel with a new one in such a way that either old or new is always available, even if the process is unexpectedly interrupted. Ever drop your cell phone and have the battery fall out?

Loading the kernel from a file system does add some complexity to the bootstrap code. The software must be able to read the storage device itself, and there must be a read-only version of the file system software required to turn a filename into disk addresses to be read. The Datalight platform simplifies this by providing standard ways to include the FlashFX Tera flash driver and a reader for the Reliance Nitro file system in boot code.

Mounting the root file system

When the Linux kernel starts, it needs access to an initial root file system containing essential system files. To access the initial root file system it must first be "mounted." In other words, the kernel must be able to access both the storage medium containing it and the software that understands the file system format. The kernel must then be able to read information describing the file system from the storage medium. This leads to a conundrum: If the kernel is in a file system that must be mounted before the kernel can be started, and the software to read the file system format must be in memory to mount the file system, what does this loading?

The answer lies in two interesting and powerful features of Linux: First, the ability to use a RAM disk as its initial root file system; and second, the ability to add new software to the kernel after it is initially loaded using kernel loadable modules (KLM).

An initial RAM disk does not need to be large, it only needs to contain a few kernel modules required to access the permanent root file system, and a small amount of code to load them. It is typically loaded by the same means as the kernel itself, from some sort of permanent storage. By way of example, a not particularly efficiently built initial RAM disk used in testing contained about 1.5 MB in files.

Running the init process

The kernel, of course, is just a means to an end: running user-mode processes that implement the desired user-visible features of a product. It gets some help in managing all those user-mode processes from one special process called “init.” The init process is the only process started directly by the kernel. By running a series of initializations scripts, it may perform important actions such as loading kernel modules and is responsible for starting (directly or indirectly) all other processes in the system.

If the system uses an initial RAM disk, the init process is found there. This can be a simple stand-alone program, or a minimal shell plus a shell script that loads modules, mounts the permanent root file system, and replaces itself with the real init process.

Extending the kernel with loadable modules

Whether or not the system uses an initial RAM disk, it's common to implement pieces of the kernel functionality as modules. This provides flexibility and convenience, since individual modules are small (tens of KB) and may be updated more easily than an entire kernel. When a module is loaded, parameters may optionally be passed to it on the command line. Since the loading process is managed from user space, it's easy to manage the module parameters with user-space tools, for example by editing a configuration file used by a script.

Introduction to the Datalight Linux Boot Loader

The Datalight boot loader project is a working example of how to load and start Linux from files stored in a Reliance partition on NAND flash managed by FlashFX. It is built for a particular hardware platform, the LogicPD MX31-lite, but can easily be adapted for other hardware. Its functions are simple, making it easy to understand and enhance. It loads the kernel and initial RAM disk from the flash, and starts the kernel. The kernel then mounts the RAM disk as a temporary root file system and runs an initialization script. This script loads the FlashFX and Reliance kernel drivers from the RAM disk, mounts the permanent root file system on the flash, and executes the init process from the new root file system.

The loader starts with the assumption that the platform hardware has been initialized sufficiently that system RAM and the NAND flash interface are usable, and that some sort of time reference is available for any delays that may be required by hardware and to time out failed operations. If a means of diagnostic output (such as a serial port) is available it is used, but this is a convenience, not a requirement. In the sample project, this initialization is expected to be accomplished by the Logic Loader (LoLo) firmware. In the general case, implementing this

minimal initialization is a simple task for either a software developer who can read hardware specifications, or a hardware designer who can write a little code.

The loader initializes FlashFX, and loads the Linux kernel and an initial RAM disk from a Reliance partition on the NAND flash. It reads these files using the Reliance Nitro Reader software that is available from Datalight.

Once the kernel and RAM disk are loaded, the loader transfers control to the kernel. After performing its own hardware initialization, the kernel starts the init process (process ID 1). The loader specifies an initialization script on the RAM disk for use as the init process.

The RAM disk contains little except for this script, a shell to interpret it, and the FlashFX and Reliance kernel modules. The script loads the modules, making the NAND flash visible as a block device. It then mounts a Reliance partition on the flash that contains the permanent root file system, changes the kernel's root file system to be this partition, and finishes by executing the real init process (which replaces the script as process ID 1).

From this point on, Linux system initialization is very conventional and may be configured in any way appropriate to the embedded system.

Analysis of the Boot Loader

The FlashFX source code is organized in a directory tree, rooted at the installation directory, with each directory containing a set of related files or subdirectories. The structure of the FlashFX installation is described in the FlashFX Tera Developer's Guide for Linux. The `projects` directory contains several preconfigured projects that serve as working examples, tests, and starting points for new projects. The boot loader is found in `projects/linux-2.6/MX31-lite/boot`. The build process for the boot loader finds most of its source code under the main FlashFX installation directory. This is adequate for the majority of FlashFX code, as it is designed to be independent of the operating environment and target hardware. However, some files are specifically intended to be customized for the hardware and software environment, and occasionally others may need to be customized too. Since the FlashFX installation directory may be shared by multiple projects (thus, those files should not be modified), customized files are placed in each project directory. In the boot loader project, they are placed in subdirectories that have the same relative path (beginning at the project directory) as the original files relative to the installation directory.

In addition to FlashFX, the loader makes use of the Reliance Nitro Reader, a separate software package available from Datalight.

The most interesting part of the boot loader is the code that is concerned specifically with booting and was newly written for this project. These files are in the project directory itself, not a subdirectory.

The top level code that loads the kernel and RAM disk and starts the kernel is in `ffxboot.c`. Like a standard C program, it has a `main()` function, which is called from some lower-level initialization code (which will be discussed later). It does all its work in 60 lines of code by taking advantage of the standard FlashFX loader and Reliance Nitro reader support, plus a few helper functions.

The process begins with a call to `FxReaderIoCreate()` (implemented in `os/loader/driver/ffxdrv.c`). This function fills in a structure (`struct sDCLREADERIO sReaderIo`) with information that will be used by the Nitro Reader functions to open and read files. It takes a device number and disk number, identifying the flash disk to access, and a partition number on the disk containing the Reliance file system with the kernel and initial RAM disk images. The correspondence between actual flash hardware and the device and disk numbers is by default set up in ffxconf.h, although it is possible to override this default mechanism and assign these numbers by other means (*see "Modifying the Project Hooks Layer" in the FlashFX Tera Developers Guide).* The partition number is an index into a standard PC-style partition table in the first block of the disk.

The reader I/O structure is used to create an instance of the Nitro Reader (the call to `NitroReaderMount()`) represented by `struct sNitroReaderInstance reader`. A pointer to this structure is then passed to all functions that access the boot volume to open and read files.

A helper function `LoadFile()` is defined locally (just after `main()`). It takes as arguments the reader structure, the address at which to load the file, and the file name, and returns the length loaded. It calls `NitroReaderOpen()` to open a file, `NitroReaderRead()` to read from it, and cleans up by closing the file with `NitroReaderClose()`.

This much of the loading process is mostly independent of the target platform. Of course, the particular device, disk, and partition must be specified, the file names must agree with what's actually stored on the flash, and the kernel and initial RAM disk are loaded at addresses that are appropriate to the platform's configuration and startup conventions. In this example, all this information is specified as literal values in the code so it is obvious; in a real product, these parameters would probably be configurable in a header file or in the makefile.

The kernel and initial RAM disk are loaded by two calls to `LoadFile()`, then the reader instance and the reader I/O instance are destroyed.

The rest of the boot process is more platform dependent, as it has to do with starting the kernel and passing the information it needs, such as hardware configuration, location of the RAM disk

image, and command line parameters. The convention for ARM Linux is to pass this information in a "tagged list," a sequence of structures. Each structure in the list begins with a header that indicates a type (for example, ATAG_MEM for memory configuration) and a size. After the header is a structure that corresponds to the indicated type (for ATAG_MEM, it indicates the starting address and size of a contiguous region of RAM). Detailed information is available in the file `Documentation/arm/Booting` in the Linux kernel source directory. Source code for the functions to build the tagged list is in `atags.c` in the project directory.

About the only feature of the boot loader that goes beyond the bare minimum is the ability to set the kernel command line at boot time, or use a default if none is set. The default command line is:

```
console=ttymxc0 init=/linuxrc root=/dev/ram0 rw noalign
```

This sets the system console to be a serial port, specifies the startup script to be run as the init process, make the RAM disk the root file system, and writable, and controls ARM alignment handling. The `init` parameter will be discussed further below.

The last step is to transfer control to the kernel. It's entered via a function call, with three arguments: the first should always be zero; the second is the machine type (`0x4d4` represents the MX31-lite); the last is a pointer to the tagged list. It's not actually ever expected to return (but there's provision for that anyway).

There is some additional code in the project that is platform-specific. It falls into two categories: DCL and FlashFX standard interfaces to platform hardware (for example, console I/O, timing); and initialization of the C runtime environment (the code that calls `main()`).

The interfaces to platform hardware are in the `os` and `dcl/os` project directories. The files in these directories are locally customized copies of the files in the corresponding directories relative to the FlashFX Tera SDK installation directory. In some cases the standard versions are adequate; only the ones that have to be tailored to the specific platform are copied to the project. For example, the loader is a single-threaded environment, so the stub versions of the DCL mutex and semaphore functions can be used, but platform specific code is needed for serial I/O and timing. These functions are not particularly interesting; details of the I/O registers may differ from platform to platform, but polled serial port I/O is about the same everywhere.

The startup code and initialization of the C runtime environment also differs across platforms, but this code illustrates some useful techniques. The basic problem that must be solved is that the boot loader must be stored in NOR flash (so it is available when the system is powered on), but is copied to RAM to run.

The entry point for the boot loader is `_start` in the assembly language file `crt0.S`. Just the bare minimum is done in assembly: the nonvolatile registers (as specified by the ARM ABI) are

saved, a new stack is established, and the boot loader is copied from NOR flash to RAM. This is sufficient to allow the C function `_cstart()` to be called with the same arguments as were passed into the assembly code entry point.

The function `_cstart()` is in the file clib.c. As the name implies, this contains minimal C library support, mainly preparing for and calling `main()`. This includes zeroing the BSS segment, where all the variables implicitly initialized to zero are located. There are also a few substitutes for functions the compiler generates references to, but which should never actually get called.

The last source file that ties all of this together is neither C nor assembly code; it is a linker script. Linker scripts are a powerful feature of the GNU linker `ld`[2]. The linker script for the boot loader specifies where the boot loader is stored in NOR flash and where it will run in RAM, and defines global symbols that can be referenced from assembly or C code.

The linker script begins with a declaration of the memory (both flash and RAM) in the system, and defines symbols to reference them (`_flashbase` and `_rambase`). It specifies which object file is placed first (`crt0.o`), where the entry point is (`_start`), and sets a value to be used for the stack size.

After this, the various sections that are generated in the object files by the compiler are assigned to segments that are given locations in both RAM and flash (this is the `>ram AT>flash`), and important locations are given names by which they can be referenced from code.

The tour of the loader source code ends with a brief mention of the configuration files `dclconf.h`, `ffxconf.h`, and `vbfconf.h`. These local copies override the default copies (in `dcl/include` and `include`, respectively) to provide custom settings for the target environment. These are all documented in the FlashFX Tera Developer's Guide for Linux and the Datalight Common Libraries Guide. Note that the settings in `ffxconf.h` and `vbfconf.h` must match the settings used to build the FlashFX device driver. Otherwise, the loader may not be able to read the flash format used by the driver.

The Initial RAM Disk

The boot loader transfers control to the kernel's entry point, and the kernel begins execution. After it has initialized the hardware and its internal data structures, it mounts a root file system and runs the first user-mode process (process ID 1). From this point on, system initialization is controlled from user space, and the kernel simply provides services.

---

[2] Documentation of linker scripts is available at http://sourceware.org/binutils/docs/ld/Scripts.html#Scripts.

The temporary root file system on the RAM disk needs to contain only the few files and device nodes that are used in the process of mounting the final root file system. The boot loader's default command line for the kernel specifies the initial process to be `/linuxrc`. This is a shell script, so there must also be a shell to interpret it. There must also be device nodes for the console and the FlashFX block device. The FlashFX kernel modules `ffxos.ko` , `flashfx.ko`, and `ffxblk.ko`, and the Reliance kernel modules `relos.ko`, `reliance.ko`, and `relfs.ko` must also be present, along with the `insmod` command to install them. A few other utilities like `mount` and `pivot_root` are also needed. As a practical matter, it is often convenient to include a few additional commands for debugging the boot process; these may be removed for a production version.

A complete directory listing of the initial RAM disk is shown in Appendix C. Note that the various commands in `/bin` are all actually symbolic links to `/bin/busybox`. BusyBox[3] combines a variety of Linux utility programs into a single executable. By limiting features, sharing code, and allowing individual commands to be included or excluded, it saves considerable space compared to separate executables and shared libraries, so it is frequently used in embedded systems.

Note: it is critical that a statically linked version of BusyBox be used for this function. Busybox will fail to load if a dynamically linked version is used. An error message similar to "Failed to execute /linuxrc. Attempting defaults..." will result from this situation.

The `/linuxrc` script is simple (and in fact could be even simpler). This is all there is to it:

```
 1   #!/bin/sh
 2
 3   set -e
 4   trap '/bin/sh' EXIT
 5
 6   insmod ffxos.ko
 7   insmod flashfx.ko
 8   insmod ffxblk.ko major=253
 9
10   insmod relos.ko
11   insmod reliance.ko
12   insmod relfs.ko
13
14   mount -r -t relfs /dev/ffx00p1 /mnt
15   cd /mnt
16   pivot_root . mnt
17   exec /mnt/bin/chroot . /sbin/init </dev/console >/dev/console 2>&1
18
19   false
```

[3] http://busybox.net/

Lines 3, 4 and 19 are present only for debugging and troubleshooting, starting an interactive shell if something goes wrong.

Lines 6-8 install the FlashFX kernel modules. The major device number is assigned explicitly to agree with the `/dev/ffx00` device node on the final root file system. Lines 10-12 similarly load the Reliance kernel modules.

Lines 14-16 mount the final root file system and make it the kernel's root file system. After the switch, the RAM disk is left mounted at /mnt. Note that the use of mnt rather than /mnt in line 16 is intentional: at the time the `pivot_root` command is executed, the intended root file system is mounted at `/mnt` and is the current directory.

Note that both the initial RAM disk and the permanent root must have a /mnt directory. The message "pivot_root: pivot_root: No such file or directory" will result if this is not the case.

Line 17 replaces the temporary init process with the real init process on the final root file system.

The file `Documentation/initrd.txt` in the Linux kernel source has an excellent description of initial RAM disks and switching the root file system.

Finishing the Linux Bootstrap Process

Once control has been transferred to the init process on the final root file system, the Linux bootstrap and initialization process can proceed in the way most appropriate to the system. A very simple embedded system might involve a single application running as the only process. More typically, the init process (such as provided by BusyBox) runs a set of initialization scripts to install modules, mount file systems, start multiple user processes, and set up networking.

Installing and Using the Boot Loader

The sample boot loader project is intended for use with the LogicPD MX31-lite, and the installation process will be described with reference to that hardware and firmware environment.

There are many possible methods for configuring the MX31-lite to use the boot loader project. The process used during development of this project involves:

- Building the kernel
- Building FlashFX and Reliance kernel modules
- Creating the initial RAM disk image

- Booting with an NFS root file system
- Initializing file systems on flash
- Installing the boot loader

Building the Kernel

The exact process for building the kernel depends on the kernel development environment being used. See your environment's documentation for details.

The rest of this description assumes that the development environment supports configuring and building a kernel to use an NFS root file system, and that a suitable root file system has been built and is available on the network.

Building FlashFX Tera and Reliance Nitro

After the kernel has been built, it's possible to build FlashFX and Reliance kernel modules for it. For each of these products, an appropriate project must be created and configured, then built. The FlashFX Tera Developer's Guide for Linux and Reliance Nitro Developer's Guide for Linux describe the process of building FlashFX and Reliance for Linux in detail.

The general process is similar for FlashFX and Reliance. The appropriate projects are in `projects/linux-2.6/MX31-lite` (FlashFX) and `projects/linux-2.6/MX31` (Reliance). Make a copy of each of these projects (including its subdirectories), edit the configuration variables in `project.mk`, and build. Depending on where you copy them to, and what kernel development environment you are using, `P_PROJDIR`, `P_PRODUCT_SRC`, `P_KERNEL_ROOT`, and `P_CROSS_COMPILE` may need to be changed.

Both the FlashFX and Reliance projects have a subdirectory called `tools`. This is where user-mode programs for these products are built. You will need to build at least the tools for Reliance in order to be able to make a Reliance file system. You may also find it convenient to build the FlashFX tools, although they are not essential. Again, it will be necessary to change some settings in `project.mk`.

The Reliance Nitro Reader is available separately from Datalight. Install it by unpacking it into its own directory. Specify the path to this directory by setting the variable `P_NITROREADER_ROOT` in the copy of `project.mk` in the boot loader project directory or on the `make` command line.

Building the Boot Loader

The boot loader project is in the FlashFX source directory `projects/linux-2.6/MX31-lite/boot`. As usual, the locations specified in `project.mk` will need to be edited.

Additionally, it is important to use compatible configuration parameters in `ffxconf.h` and `vbfconf.h` for the FlashFX driver and the boot loader.

It is also very important to check the region of the NOR flash to be used to hold the boot loader. This is specified in the linker script `ffxboot.lds` (described in *Analysis of the Boot Loader*). The boot loader will be programmed to NOR flash at the specified address. If this conflicts with flash that is used by the firmware, the board may become completely unbootable and need to be recovered through some means like a JTAG interface. Needless to say, this is inconvenient. The address configured in the project in this release worked with the MX31-lite that was used to develop this project, but that's no guarantee that it will work with a different revision of the board or firmware.

Once the configuration is set, the boot loader is built in the `boot` subdirectory by simply typing `make`. This will produce an ARM ELF file called `ffxboot`. Note that it is possible to build a debug version of this with `make B_DEBUG=1`. This may require increasing the size of the flash region allocated in the linker script (so read the previous paragraph again).

Creating the Initial RAM Disk Image

Several techniques for creating a RAM Disk image are discussed in detail in the Linux kernel source file `Documentation/initrd.txt`.

An example listing of the contents of the RAM disk image is given in Appendix C. Details of what must be included will vary.

A convenient way to experiment with RAM disk contents is to create a directory tree that will be copied to the RAM disk image. The following shell script illustrates the process. It is meant to be run on the development host system, which also acts as the NFS server for the target system. The NFS root file system is exported from `/tftpboot/mx31/rootfs`, and the subdirectory `test` contains the contents destined for the RAM disk. The image is created in what will appear as `/tmp` on the target system.

```
 1   #! /bin/sh
 2
 3   set -e
 4
 5   cd /tftpboot/mx31/rootfs/initrd.master
 6
 7   dd if=/dev/zero of=../tmp/initrd bs=1k count=1500
 8   /sbin/mkfs -t ext2 -F -N 160 ../tmp/initrd
 9   mount -o loop ../tmp/initrd /mnt/spare/
10   tar -cf - . | (cd /mnt/spare/ && tar xf -)
11   df /mnt/spare
12   umount /mnt/spare/
```

```
13    gzip ../tmp/initrd
```

Note that this script must be run as root.

Installing Using an NFS Root File System

The most convenient way to prepare the flash memory and install the kernel and initial RAM disk is by booting the target system using an NFS root file system. The following description assumes that a root file system has been prepared and exported, and the target system has been booted with it as the root file system. The previous step (preparing the initial RAM disk image) must have been completed as described. The FlashFX and Reliance kernel modules and utilities that were built earlier must be placed somewhere on the NFS exported file system so they can be used on the target system.

The first step is to initialize the NAND flash. Assuming it has never been used before, it will need to be formatted at two levels. First the Bad Block Manager (BBM) data structures must be created to record any bad blocks marked by the flash manufacturer. Then the Variable Block Format (VBF) data structures are created to enable simulating a random-access block device. Both of these functions may be enabled when loading the FlashFX kernel modules:

```
insmod ffxos.ko bbmformat=0,1
insmod flashfx.ko
insmod ffxblk.ko format=0,1
```

Depending on the size of the NAND flash, this may take a few seconds to a few minutes.

Note that if the flash has previously been used by other software, that software should be used to erase the flash (except for factory bad blocks). Otherwise BBM formatting may fail because the other software's format may be interpreted as bad block marks.

Once the formatting has been completed, the FlashFX driver will make the flash available as a block device. In order to access the block device, device nodes must be created to refer to it. If the target system is configured to use `udev`, this will occur automatically, and `/dev/ffx00` will appear as if by magic a few seconds after the driver completes its initialization. In many embedded systems, the configuration does not change at run time, so udev is not used. In this case, the device node must be created manually, and this requires knowing the major device number FlashFX is using.

In the `insmod` command line given above, no major number was specified, so the driver was assigned one automatically. This may be determined by examining `/proc/devices`. In its "Block devices:" section it will have a line like:

```
253 ffx
```

Use this number as the major number to create the block device node for the whole disk:

```
mknod /dev/ffx00 b 253 0
```

Now that there is a node for the whole disk you can use the `fdisk` command to create the desired partitions. In the default configuration of the boot loader, the first one is expected to be a Reliance Nitro partition and contain the kernel and RAM disk. For convenience during updates, make it big enough to hold two copies of each. Optionally, you may create a second partition for the root file system, and mount the first partition as `/boot`. Depending on your intended use, you may want more partitions.

Again, if the system includes udev, device nodes will be created automatically for the partitions, otherwise manually create a node for each partition that was created with fdisk:

```
mknod /dev/ffx00p1 b 253 1
mknod /dev/ffx00p2 b 253 2
```

Appropriate file systems must be created on each partition. To create a Reliance file system:

```
mkfs.relfs -k /dev/ffx00p1
```

The –k switch means to skip the scan for bad blocks (there aren't any because BBM already mapped them out, it would just waste time).

The root file system may be populated with files copied from the NFS file system. At a minimum, it needs the executable that will run as the init process (typically /sbin/init) along with any shared libraries it is linked to, and device nodes for all the devices that will be accessed. Generally, there will be many more files than this. A convenient (and self-documenting) way to copy the desired files is by using `cpio` in copy-pass mode, using a text file that lists the exact files and directories to be copied as input to `cpio`.

Installing the Boot Loader

The LoLo firmware supports downloading the ELF image and programming it into NOR flash. It gets the flash address from the ELF header (where it was set by the linker script).

Place `ffxboot` in the TFTP directory, and download it as normal, then issue the `burn` command. The `load` and `burn` commands are documented in the *LogicLoader™ Command Description Manual* (Logic PN 1002184) and *LogicLoader™ User's Manual* (Logic PN 70000016).

Using the Boot Loader

At this point, all the pieces should be in place. The boot loader is activated with the `exec` command, giving the address where it was stored in NOR flash (as specified in the linker script). For example:

```
exec 0xa0100000
```

The `exec` command supports passing a command line to the boot loader, which simply passes it on to the kernel unchanged. This is especially useful for troubleshooting problems with the `/linuxrc` script. Pass the default command line, except change it to include `init=/bin/sh`.

Concluding Thoughts

At the beginning of this paper, we made mention of the process appearing complicated, convoluted, and arcane. Having laid out the steps and followed them, it should now be apparent that booting Linux from NAND does not have to be any of these things. The undertaking is also made much easier by using a purposefully-designed and well-documented flash file system combined with help from a professional support organization. Keep checking back with us for updates at www.Datalight.com as we continue to make improvements in the functionality and performance of our products.

Appendix A: List of project files (with brief descriptions)

`atags.c` – functions to create tagged lists for kernel startup
`atags.h` – interface to `atags.c`
`clib.c` – C runtime initialization
`crt0.S` – assembly language entry point
`dclconf.h` – configuration for DCL
`dcl/os/loader/services/osdate.c` – read system clock
`dcl/os/loader/services/osfile.c` – stubs, file I/O not supported
`dcl/os/loader/services/osfileio.c` – stubs, file I/O not supported
`dcl/os/loader/services/osinput.c` – console serial port input
`dcl/os/loader/services/osmem.c` – heap allocation
`dcl/os/loader/services/osoutput.c` – console serial port output
`dcl/os/loader/services/ossleep.c` – millisecond resolution delays
`dcl/os/loader/services/ostick.c` – clock tick for millisecond delays
`ffxboot.c` – boot loader main function
`ffxboot.lds` – linker script for building a flash image of the boot loader
`ffxconf.h` – general FlashFX configuration

`Makefile` – general project make file
`mx31io.h` – memory-mapped I/O interface
`mx31reg.h` – memory-mapped I/O addresses
`os/loader/hooks/fhmx31.c` – setup for MX31 NAND controller
`os/loader/services/oedelay.c` – microsecond resolution delays
`project.h` – internal interfaces between project modules
`project.mk` – make file customization

Appendix B: Glossary

DCL, FIM, NTM,...

Appendix C: Initial RAM disk contents

The following directory listing shows the full contents of the initial RAM disk used during development and testing of the boot loader. Note that not everything in this list is essential; many items were left in for convenience troubleshooting any problems.

```
.:
total 740
drwxr-xr-x. 2 root root   4096 2008-07-11 15:22 bin
drwxr-xr-x. 6 root root   4096 2008-07-10 14:50 dev
drwxr-xr-x. 2 root root   4096 2008-07-11 14:02 etc
-rw-r--r--. 1 root root  21885 2010-04-07 16:44 ffxblk.ko
-rw-r--r--. 1 root root  21073 2010-04-07 16:44 ffxos.ko
-rw-r--r--. 1 root root 419000 2010-04-07 16:44 flashfx.ko
-rwxr-xr-x. 1 root root    289 2010-04-07 16:44 linuxrc
drwxr-xr-x. 2 root root   4096 2008-07-09 17:41 mnt
drwxr-xr-x. 2 root root   4096 2008-07-09 17:41 proc
-rw-r--r--. 1 root root  27355 2010-04-07 16:45 relfs.ko
-rw-r--r--. 1 root root 142732 2010-04-07 16:45 reliance.ko
-rw-r--r--. 1 root root  21235 2010-04-07 16:45 relos.ko
drwxr-xr-x. 2 root root   4096 2008-07-09 17:41 sys

./bin:
total 908
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 [ -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 [[ -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 ash -> busybox
-rwsr-xr-x. 1 root root 763864 2008-07-11 11:15 busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 cat -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 chroot -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 cp -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 echo -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 false -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 freeramdisk -> busybox
```

```
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 hostname -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 ifconfig -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 insmod -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 ipaddr -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 iplink -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 ln -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 ls -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 lsmod -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 mkdir -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 mknod -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 mount -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 mv -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 pivot_root -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 pwd -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 rdate -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 rm -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 rmdir -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 rmmod -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 run-parts -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 setconsole -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 sh -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 sleep -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 stty -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 switch_root -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 sync -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 test -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 true -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 tty -> busybox
lrwxrwxrwx. 1 root root      7 2010-04-07 16:42 umount -> busybox

./dev:
total 356
crw-rw-rw-. 1 root root   5,   1 2007-05-11 15:12 console
brw-------. 1 root root 253,   0 2008-06-20 14:04 ffx00
brw-------. 1 root root 253,   1 2008-06-20 14:06 ffx00p1
brw-------. 1 root root 253,   2 2008-06-20 14:06 ffx00p2
brw-------. 1 root root 253,   3 2008-06-20 14:06 ffx00p3
prw-------. 1 root root        0 2008-06-20 13:38 initctl
drwxr-xr-x. 2 root root     4096 2007-05-11 15:12 input
crw-r-----. 1 root root   1,   2 2007-05-11 15:12 kmem
lrwxrwxrwx. 1 root root       12 2010-04-07 16:42 log -> /var/tmp/log
brw-r-----. 1 root root   7,   0 2007-05-11 15:12 loop0
brw-r-----. 1 root root   7,   1 2007-05-11 15:12 loop1
crw-r-----. 1 root root   1,   1 2007-05-11 15:12 mem
lrwxrwxrwx. 1 root root       12 2010-04-07 16:42 mtab -> /proc/mounts
drwxr-xr-x. 2 root root     4096 2007-05-11 15:12 net
crw-rw-rw-. 1 root root   1,   3 2007-05-11 15:12 null
crw-rw-rw-. 1 root root 108,   0 2007-05-11 15:12 ppp
crw-rw-rw-. 1 root root  10,   1 2007-05-11 15:12 psaux
crw-rw-rw-. 1 root root   5,   2 2007-05-11 15:12 ptmx
```

```
drwxr-xr-x. 2 root root     4096 2007-05-11 15:12 pts
crw-rw-rw-. 1 root root   2,   0 2007-05-11 15:12 ptyp0
crw-rw-rw-. 1 root root   2,   1 2007-05-11 15:12 ptyp1
crw-rw-rw-. 1 root root   2,   2 2007-05-11 15:12 ptyp2
crw-rw-rw-. 1 root root   2,   3 2007-05-11 15:12 ptyp3
crw-rw-rw-. 1 root root   2,   4 2007-05-11 15:12 ptyp4
crw-rw-rw-. 1 root root   2,   5 2007-05-11 15:12 ptyp5
crw-rw-rw-. 1 root root   2,   6 2007-05-11 15:12 ptyp6
crw-rw-rw-. 1 root root   2,   7 2007-05-11 15:12 ptyp7
crw-rw-rw-. 1 root root   2,   8 2007-05-11 15:12 ptyp8
crw-rw-rw-. 1 root root   2,   9 2007-05-11 15:12 ptyp9
brw-r-----. 1 root root   1,   1 2007-05-11 15:12 ram
brw-r-----. 1 root root   1,   0 2007-05-11 15:12 ram0
brw-r-----. 1 root root   1,   1 2007-05-11 15:12 ram1
brw-r-----. 1 root root   1,   2 2007-05-11 15:12 ram2
brw-r-----. 1 root root   1,   3 2007-05-11 15:12 ram3
crw-rw-rw-. 1 root root   1,   8 2007-05-11 15:12 random
crw-r-----. 1 root root  10, 135 2007-05-11 15:12 rtc
drwxr-xr-x. 2 root root     4096 2007-05-11 15:12 shm
crw-rw-rw-. 1 root root  14,   6 2007-05-11 15:12 sndstat
crw-r--r--. 1 root root  10,  16 2007-05-11 15:12 ts
crw-rw-rw-. 1 root root   5,   0 2007-05-11 15:12 tty
crw-rw-rw-. 1 root root   4,   0 2007-05-11 15:12 tty0
crw-rw-rw-. 1 root root   4,   1 2007-05-11 15:12 tty1
crw-rw-rw-. 1 root root   4,   2 2007-05-11 15:12 tty2
crw-rw-rw-. 1 root root   4,   3 2007-05-11 15:12 tty3
crw-rw-rw-. 1 root root   4,   4 2007-05-11 15:12 tty4
crw-rw-rw-. 1 root root   4,   5 2007-05-11 15:12 tty5
crw-rw-rw-. 1 root root   4,   6 2007-05-11 15:12 tty6
crw-rw-rw-. 1 root root   4,   7 2007-05-11 15:12 tty7
crw-rw-rw-. 1 root root 204,  16 2007-05-11 15:12 ttyAM0
crw-rw-rw-. 1 root root 204,  17 2007-05-11 15:12 ttyAM1
crw-rw-rw-. 1 root root 204,  18 2007-05-11 15:12 ttyAM2
crw-rw-rw-. 1 root root 204,  19 2007-05-11 15:12 ttyAM3
crw-rw-rw-. 1 root root 204,  46 2007-05-11 15:12 ttyCPM0
crw-rw-rw-. 1 root root 204,  47 2007-05-11 15:12 ttyCPM1
crw-rw----. 1 root root 207,  16 2007-05-11 15:12 ttymxc0
crw-rw-rw-. 1 root root   3,   0 2007-05-11 15:12 ttyp0
crw-rw-rw-. 1 root root  57,   0 2007-05-11 15:12 ttyP0
crw-rw-rw-. 1 root root   3,   1 2007-05-11 15:12 ttyp1
crw-rw-rw-. 1 root root  57,   1 2007-05-11 15:12 ttyP1
crw-rw-rw-. 1 root root   3,   2 2007-05-11 15:12 ttyp2
crw-rw-rw-. 1 root root  57,   2 2007-05-11 15:12 ttyP2
crw-rw-rw-. 1 root root   3,   3 2007-05-11 15:12 ttyp3
crw-rw-rw-. 1 root root  57,   3 2007-05-11 15:12 ttyP3
crw-rw-rw-. 1 root root   3,   4 2007-05-11 15:12 ttyp4
crw-rw-rw-. 1 root root   3,   5 2007-05-11 15:12 ttyp5
crw-rw-rw-. 1 root root   3,   6 2007-05-11 15:12 ttyp6
crw-rw-rw-. 1 root root   3,   7 2007-05-11 15:12 ttyp7
crw-rw-rw-. 1 root root   3,   8 2007-05-11 15:12 ttyp8
```

```
crw-rw-rw-. 1 root root   3,   9 2007-05-11 15:12 ttyp9
crw-rw-rw-. 1 root root   4,  64 2007-05-11 15:12 ttyS0
crw-rw-rw-. 1 root root   4,  65 2007-05-11 15:12 ttyS1
crw-rw-rw-. 1 root root   4,  66 2007-05-11 15:12 ttyS2
crw-rw-rw-. 1 root root   4,  67 2007-05-11 15:12 ttyS3
brw-r-----. 1 root root  98,   0 2007-05-11 15:12 ubda
brw-r-----. 1 root root  98,   1 2007-05-11 15:12 ubda1
brw-r-----. 1 root root  98,   2 2007-05-11 15:12 ubda2
brw-r-----. 1 root root  98,   3 2007-05-11 15:12 ubda3
brw-r-----. 1 root root  98,   4 2007-05-11 15:12 ubda4
brw-r-----. 1 root root  98,   5 2007-05-11 15:12 ubda5
brw-r-----. 1 root root  98,   6 2007-05-11 15:12 ubda6
brw-r-----. 1 root root  98,   7 2007-05-11 15:12 ubda7
brw-r-----. 1 root root  98,   8 2007-05-11 15:12 ubda8
crw-rw-rw-. 1 root root   1,   9 2007-05-11 15:12 urandom
crw-r--r--. 1 root root  10, 130 2007-05-11 15:12 watchdog
crw-rw-rw-. 1 root root   1,   5 2007-05-11 15:12 zero

./dev/input:
total 0

./dev/net:
total 0

./dev/pts:
total 0

./dev/shm:
total 0

./etc:
total 40
-rw-r--r--. 1 root root  50 2008-06-20 13:30 busybox.conf
-rw-r--r--. 1 root root 709 2004-08-13 07:57 inputrc
-rw-r--r--. 1 root root  37 2006-08-22 12:09 profile
-rw-r--r--. 1 root root 178 2006-08-22 12:09 securetty
-rw-r--r--. 1 root root  93 2002-07-19 10:04 shells

./mnt:
total 0

./proc:
total 0

./sys:
total 0
```